

AD-A231 506

15  
DTIC FILE COPY

UMIACS-TR-90-139  
CS-TR-2562

November 1990

**Parallel Implementation of the *hp*-version of the Finite Element Method on a Shared-Memory Architecture**

I. Babuška<sup>1</sup>  
H. C. Elman<sup>2</sup>  
K. Markley<sup>3</sup>

University of Maryland  
Institute for Advanced Computer Studies Report

DTIC  
S ELECTE FEB 05 1991 D  
D

**Abstract**

We study the costs incurred by an implementation of the *hp*-version of the finite element for solving two-dimensional elliptic partial differential equations on a shared-memory parallel computer. For a collection of benchmark problems, we systematically examine the costs in CPU time of various individual subtasks performed by the finite element solver, including construction of local stiffness matrices, elimination of unknowns associated with element interiors, and global solution on element interfaces by a preconditioned conjugate gradient method. Our general observations are that the costs of the "naturally" parallel computations associated with local elements are significantly higher than any global computations, so that the latter do not represent a significant bottleneck to parallel efficiency. However, memory conflicts place some limitations on the sizes or number of local problems that can be handled efficiently in parallel.

**Abbreviated Title.** Parallel implementation of *hp*-version.

**Key words.** Finite element, *hp*-version, parallel, shared memory, domain decomposition.

**AMS (MOS) subject classification.** Primary: 65F10, 65N20, 65W05.

<sup>1</sup>Department of Mathematics and Institute for Physical Science and Technology, University of Maryland, College Park, MD 20742. The work of this author was supported by the U. S. Office of Naval Research under contract N00014-90-J-1030, and by the National Science Foundation under grant CCR-88-20279.

<sup>2</sup>Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. The work of this author was supported by the U. S. Army Research Office under grant DAAL-0389-K-0016, and by the National Science Foundation under grants ASC-8958544 and DMS-8607478. Computer time was provided by the Advanced Computing Research Facility at Argonne National Laboratory.

<sup>3</sup>Rice University, Houston, TX 77001.

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

91 2 5 033

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER UMIACS-TR-90-139/CS-TR-2562	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Parallel Implementation of the hp-version of the Finite Element Method on a Shared-Memory Architecture		5. TYPE OF REPORT & PERIOD COVERED Final life of contract
7. AUTHOR(s) <sup>1</sup> I. Babuska, <sup>2</sup> H. C. Elman, K. Markley <sup>2</sup> ARO DAAL-0389-K-0016		8. CONTRACT OR GRANT NUMBER(S) ONR N00014-90-J-1030 NSF CCR-88-20279 NSF ASC-8958544, DMS-8607478
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Mathematics and Institute for Physical Science and Technology University of Maryland - College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, VA 22217		12. REPORT DATE November 1990
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES 28
		16. SECURITY CLASS. (of this report)
		18a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release: distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 30, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Finite element, hp-version, parallel, shared memory, domain decomposition		
20. ABSTRACT We study the costs incurred by an implementation of the hp-version of the finite element for solving two-dimensional elliptic partial differential equations on a shared-memory parallel computer. For a collection of benchmark problems, we systematically examine the costs in CPU time of various individual subtasks performed by the finite element solver, including construction of local stiffness matrices, elimination of unknowns associated with element interiors, and global solution on element interfaces by a preconditioned conjugate gradient method. Our general observations are that the costs of the "naturally" parallel computations associated with local elements are significantly higher than any global computations, so that the latter do not represent a significant bottleneck to parallel efficiency. However, memory conflicts place some limitations on the sizes or number of local problems that can be handled efficiently in parallel.		

**1. Introduction.** The finite element method is a standard computational tool for solving partial differential equations arising from engineering analysis. Variants include the standard *h*-version, which uses low-order basis functions and achieves accuracy by refining meshes [12]; the *p*-version, which uses a fixed mesh and achieves accuracy by increasing the order of the basis functions; and the *hp*-version, which combines these two approaches. See [7] for a survey and comprehensive list of references on the *p*- and *hp*-versions. For the first and last of these techniques, which divide domains into local elements and compute associated local stiffness operators, a large component of the required computations can be implemented very naturally on parallel architectures. In particular, for the *h*-version, *domain decomposition methods* (e.g. [8],[9],[10],[11],[23],[24],[28]) group collections of elements into subdomains, or *super-elements*; construction of all local stiffness operators associated with super-elements, as well as elimination of degrees of freedom internal to super-elements, are independent of one another, so that they can be performed in parallel on separate processors. For the *p* and *hp*-version, one can think of the space of high-order basis functions in each element as analogous to the grouping of *h*-elements into super-elements. Then, just as for domain decomposition methods, construction of local operators and partial elimination can be done in a natural way on independent processors. A combination of these points of view, with multiple high-order elements collected into super-elements, is also possible.

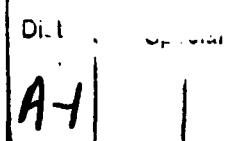
After the fully local computations have been performed, the result is a subproblem with unknowns on super-element interfaces. If an iterative method such as the conjugate gradient method (CG) is used to solve this subproblem, then much of the required computation is also local, so that there is a large amount of natural parallelism. However, these computations entail some interaction across super-element interfaces, and, in addition, CG requires some global computations. Moreover, convergence of such methods is often significantly accelerated by some type of global preconditioner [3],[8],[9],[10],[11], which may be less natural to implement in parallel. The effects of both super-element interactions and global operations on overall performance on parallel architectures is not well-understood.

In this paper, we describe the results of an experimental study of an implementation of the *hp*-version of the finite element method for solving two-dimensional linear elliptic problems on a shared-memory parallel computer. We examined the computational costs of the various subtasks required by the *hp*-method, including:

- construction of local stiffness matrices;
- partial elimination of unknowns associated with purely local elements;
- CG iteration;
- preconditioning derived from low-order elements.

Our goals were to determine how efficiently such computations can be done on parallel architectures, and what bottlenecks may exist that limit efficiency. Some particular issues considered were:

- the relative costs of the various individual subtasks;
- the effects of global operations, especially preconditioning, on overall performance and parallel efficiency;



- the overhead of using unassembled local matrices to perform the matrix-vector products required by CG; and
- whether there are any limitations associated with the "natural" subdivision of problems based on independent elements.

Our tests were performed on an Alliant FX/8, an eight-processor shared-memory computer with a fast cache memory and vector processors. In addition to examining the general issues of parallel implementation, we also considered the effects of the latter two architectural features. In general, we found that the dominant costs come from the local computations, especially the construction of local stiffness matrices, and that global computations required for fast convergence do not represent a significant bottleneck. There are some drawbacks to having local computations that operate on large sets of data, though, which appear to be architecture-related, derived from inefficient data movement for parallel processing.

An outline of the paper is as follows. In §2, we present the continuous model problem used for experiments, and we describe the *hp*-version of the finite element method used for the discretization. In §3, we give a high level description of the solution algorithm and a detailed description of our implementation. §4 contains the main results of the paper, a series of experimental results for a set of benchmark problems. These include overviews of iteration counts and CPU times, as well as several refinements of timing statistics showing where computational efforts are spent, as well as analyses of the effects of synchronization and vectorization. Finally, in §5, we summarize our observations and discuss their implications for computations on other classes of problems and parallel computers.

**2. The Model Problem and its Finite Element Solution.** Consider the model problem

$$-\left(\frac{\partial}{\partial x} a \frac{\partial u}{\partial x} + \frac{\partial}{\partial y} b \frac{\partial u}{\partial y}\right) = f \text{ on } \Omega, \quad (1)$$

$$u = g_d \text{ on } \Gamma_D, \quad \frac{\partial u}{\partial n_c} = g_n \text{ on } \Gamma_N. \quad (2)$$

Here,  $\Omega \subset \mathbf{R}^2$  is a bounded domain with piecewise smooth (e.g. polygonal) boundary,  $\Gamma = \Gamma_D \cup \Gamma_N$  is the boundary of  $\Omega$ ,  $a$ ,  $b$ ,  $f$ ,  $g_d$  and  $g_n$  are functions that satisfy the usual conditions guaranteeing existence and uniqueness of the solution, and  $n_c$  is the conormal. We are interested in the weak solution of (1) – (2), i.e.  $u \in H_D^1(\Omega) \equiv \{u \in H^1(\Omega) | u = 0 \text{ on } \Gamma_D\}$  such that

$$B(u, v) \equiv \int_{\Omega} a \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + b \frac{\partial u}{\partial y} \frac{\partial v}{\partial y} dx dy = \int_{\Omega} f v dx dy + \int_{\Gamma_N} g_n v ds \equiv F(v)$$

holds for any  $v \in H_D^1(\Omega)$ .  $H^1(\Omega)$  denotes the usual Sobolev space.

We now give a general description of the *hp*-version of the finite element discretization of (1) – (2). Let  $\mathcal{P} = \{\Omega^i\}$  denote a partitioning of  $\Omega$  into open subdomains such that

$$\bar{\Omega} = \cup \bar{\Omega}^i$$

(where  $\bar{\Omega}$  denotes the closure of  $\Omega$ ). Assume that  $\Omega^i$  is a curvilinear polygon, typically a triangle or quadrilateral. Let

$$\Gamma^i = \cup_{j=1}^{m_i} \Gamma_j^i$$

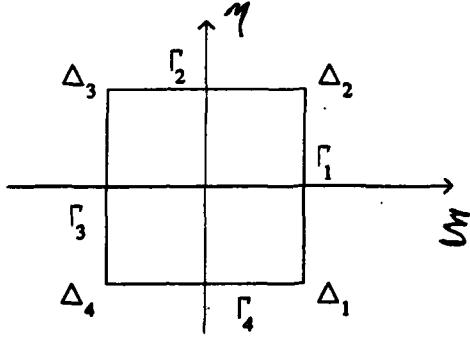


Figure 1: The standard element  $\mathcal{E}$ .

denote the set of (curved) open sides of  $\Omega^i$ , and let

$$\Delta^i = \cup_{j=1}^{m_i} \Delta_j^i$$

denote its vertices. Assume that either

1.  $\bar{\Omega}^i \cap \bar{\Omega}^j = \Gamma_k^i = \Gamma_l^j$ , i.e.  $\bar{\Omega}^i$  and  $\bar{\Omega}^j$  have common entire sides; or
2.  $\bar{\Omega}^i \cap \bar{\Omega}^j = \Delta_k^i = \Delta_l^j$ , i.e.  $\bar{\Omega}^i$  and  $\bar{\Omega}^j$  have one vertex in common; or
3.  $\bar{\Omega}^i \cap \bar{\Omega}^j = \emptyset$ .

The set  $(\cup_{i,j} \Gamma_j^i) \cup (\cup_{i,j} \Delta_j^i)$  will be denoted the *frame* of the partitioning  $\mathcal{P}$ .

On every  $\Omega^i \in \mathcal{P}$ , we use a set of linearly independent functions  $\Phi_j^i \in H^1(\Omega^i)$ ,  $j = 1, \dots, \rho_i$ , called *shape functions*, which are divided into three categories:

*Internal shape functions:*  $\mathcal{I} \subset \{\Phi^{(T)} \in H^1(\Omega^i) \mid \Phi^{(T)} = 0 \text{ on } \Gamma^i\}$ .

*Side shape functions:*  $\mathcal{S} \subset \{\Phi^{(S,j)} \in H^1(\Omega^i) \mid \Phi^{(S,j)} = 0 \text{ on } \Gamma^i - \Gamma_j^i\}$ .

*Nodal shape functions:*  $\mathcal{N} \subset \{\Phi^{(N,j)} \in H^1(\Omega^i) \mid \Phi^{(N,j)} = 0 \text{ on } \Delta_k^i, k \neq j\}$ .

The following typical examples will be used in the sequel. Let  $\mathcal{E} = (-1, 1) \times (-1, 1)$  be called the *standard element*. The nodes and sides of  $\mathcal{E}$  are given by

$$\Delta_1 = (1, -1), \Delta_2 = (1, 1), \Delta_3 = (-1, 1), \Delta_4 = (-1, -1),$$

$$\begin{aligned} \Gamma_1 &= \{(\xi, \eta) \mid \xi = 1, |\eta| < 1\}, & \Gamma_2 &= \{(\xi, \eta) \mid |\xi| < 1, \eta = 1\}, \\ \Gamma_3 &= \{(\xi, \eta) \mid \xi = -1, |\eta| < 1\}, & \Gamma_4 &= \{(\xi, \eta) \mid |\xi| < 1, \eta = -1\}. \end{aligned}$$

(See Fig. 1.) Let

$$\phi_j(\xi) = \sqrt{\frac{2j-1}{2}} \int_{-1}^{\xi} P_{j-1}(t) dt, \quad j \geq 2, \quad (3)$$

where  $P_j(t)$  is the Legendre polynomial of degree  $j$  [13]. Thus,  $\phi_j(\xi)$  is a polynomial of degree  $j$  and  $\phi_j(\pm 1) = 0$ . Two spaces,  $\mathcal{Q}(p)$  and  $\mathcal{Q}'(p)$ , are defined as the span of the following shape functions on  $\mathcal{E}$ :

*Internal shape functions:* For  $\mathcal{Q}(p)$ ,

$$\Phi_{jk}^{(T)}(\xi, \eta) = \phi_j(\xi) \phi_k(\eta), \quad 2 \leq j, k \leq p;$$

and for  $\mathcal{Q}'(p)$ ,

$$\Phi_{jk}^{(I)}(\xi, \eta) = \phi_j(\xi)\phi_k(\eta), \quad j, k \geq 2, \quad j + k \leq p.$$

Hence, for  $\mathcal{Q}(p)$ , there are  $(p - 1)^2$  internal shape functions, and for  $\mathcal{Q}'(p)$  there are  $(p - 2)(p - 3)/2$  internal shape functions when  $p \geq 4$ , and none when  $p < 4$ .

*Side shape functions:* For both  $\mathcal{Q}(p)$  and  $\mathcal{Q}'(p)$ , the side shape functions are given by

$$\begin{aligned}\Phi_j^{(S,1)}(\xi, \eta) &= \left(\frac{\xi+1}{2}\right)\phi_j(\eta), & \Phi_j^{(S,2)}(\xi, \eta) &= \phi_j(\xi)\left(\frac{\eta+1}{2}\right), \\ \Phi_j^{(S,3)}(\xi, \eta) &= \left(\frac{-\xi+1}{2}\right)\phi_j(\eta), & \Phi_j^{(S,4)}(\xi, \eta) &= \phi_j(\xi)\left(\frac{-\eta+1}{2}\right),\end{aligned}\quad j = 2, \dots, p.$$

Thus, there is a total of  $4(p - 1)$  side shape functions.

*Nodal shape functions.* For both  $\mathcal{Q}(p)$  and  $\mathcal{Q}'(p)$ , the four nodal shape functions are given by

$$\begin{aligned}\Phi^{(N,1)}(\xi, \eta) &= \left(\frac{\xi+1}{2}\right)\left(\frac{-\eta+1}{2}\right), & \Phi^{(N,2)}(\xi, \eta) &= \left(\frac{\xi+1}{2}\right)\left(\frac{\eta+1}{2}\right), \\ \Phi^{(N,3)}(\xi, \eta) &= \left(\frac{-\xi+1}{2}\right)\left(\frac{\eta+1}{2}\right), & \Phi^{(N,4)}(\xi, \eta) &= \left(\frac{-\xi+1}{2}\right)\left(\frac{-\eta+1}{2}\right).\end{aligned}$$

Here,  $\mathcal{Q}(p)$  contains all polynomials of degree  $p$  in each variable, and  $\mathcal{Q}'(p)$  both contain all polynomials of total degree  $p$ . For some choices of  $p$ , internal or side shape functions are not present, and the method can reduce to the standard  $h$ -version. Also, the spaces  $\mathcal{Q}(p)$  and  $\mathcal{Q}'(p)$  could be defined as the span of some other shape functions. For example,  $\mathcal{Q}(p)$  is the span of all functions of the form  $f_j(\xi)f_k(\eta)$  where  $\{f_j(\xi)\}$  are Lagrange polynomials with interpolation points chosen to be the  $(p + 1)$  Gauss-Lobatto quadrature points [25]. Similar sets of standard shape functions can be defined on a triangular element.

Assume that  $T_i(\xi, \eta)$  is a mapping of the standard element  $\mathcal{E}$  onto  $\Omega^i$ . Let  $\mathcal{Q}^*$  denote either  $\mathcal{Q}(p)$  or  $\mathcal{Q}'(p)$ , and let

$$\mathcal{V} = \{u \in H_D^1(\Omega) \mid u|_{\Omega^i}(x, y) = v(T_i^{-1}(x, y)), v \in \mathcal{Q}^*\}.$$

We impose on  $T_i$  the usual conditions of the finite element method, e.g. if  $\Omega^i$  is a parallelogram then  $T_i$  is a linear mapping. Thus, the basis functions of  $\mathcal{V}$  can easily be constructed using the three categories of standard shape functions. The finite element solution  $u_{FE} \in \mathcal{V}$  is defined by

$$B(u_{FE}, v) = F(v) \text{ for all } v \in \mathcal{V}. \quad (4)$$

Condition (4) uniquely defines  $u_{FE}$  except when  $\Gamma_D = \emptyset$ , in which case  $u_{FE}$  is determined up to a constant.

Accuracy of  $u_{FE}$  is achieved either by increasing the degree  $p$  of the shape functions, or by refining the partitioning  $\mathcal{P}$ . Consider the case where  $\mathcal{P}$  partitions a rectangular domain  $\Omega$  into an  $m \times n$  rectangular grid composed of squares with side  $h$ . The following results contain typical error bounds for the finite element solution in the energy norm

$$\|u - u_{FE}\|_E \equiv B(u - u_{FE}, u - u_{FE})^{1/2} = \inf_{v \in \mathcal{V}} B(u - v, u - v)^{1/2}.$$

See [7] and references therein for further details.

**Theorem.** (1) Suppose  $h$  is such that  $c_1 h^{-1} \leq m, n \leq c_2 h^{-1}$  and  $u \in H^k(\Omega)$ . Then

$$\|u - u_{FE}\|_E \leq C \frac{h^\alpha}{p^{k-1}} \|u\|_{H^k},$$

where  $\alpha = \min\{k-1, p\}$  and  $C$  depends on  $k, c_1, c_2$  and the discretization ( $Q(p)$  or  $Q'(p)$ ), but is independent of  $u, h$  and  $p$ .

(2) If  $u$  is analytic in  $\bar{\Omega}$ , then for any fixed  $h > 0$ ,

$$\|u - u_{FE}\|_E \leq D e^{-\beta p},$$

where  $D$  depends on  $u$  and  $h$  and  $\beta > 0$  depends on the region in which  $u$  is analytic.

Thus, for very smooth problems, the  $p$  and  $hp$  finite element solution displays exponential convergence.

For our investigation, we will restrict our attention to the case where  $\Omega = (-1, 1) \times (-1, 1)$ ;  $\mathcal{P}$  partitions  $\Omega$  into a uniform  $n \times n$  grid;  $T_i$  is a bilinear mapping; and  $f = 0, \Gamma_D = \emptyset$ . We will consider the constant coefficient case  $a = b = 1$ . To ensure a unique solution, we will constrain the solution at the corners of  $\partial\Omega$ .

**3. The Solution Algorithm and its Implementation.** Formal specification of the finite element solution  $u_{FE}$  as a linear combination of the basis functions of  $\mathcal{V}$  leads to a system of linear equations

$$S\alpha = s,$$

where  $S$  is the global stiffness matrix and  $\alpha$  is the vector of coefficients of the basis functions. In this section, we present the algorithm used to solve this problem and describe the details of our implementation.

**3.1. The Solution Algorithm.** Conceptually, the algorithm can be divided into four steps.

1. *Construction of the local stiffness matrices.* The global matrix has the form

$$S = \sum_i S_i,$$

where  $S_i$  is the local stiffness matrix associated with  $\Omega^i$ . Formally,  $S_i$  is a large, sparse matrix with nonzero entries determined from shape functions in  $\Omega^i$ . In the following,  $S_i$  will also be identified with the local matrix of order  $\rho_i$  given by the Gramm matrix  $[B_{\Omega^i}(u, v)]$ , where

$$B_{\Omega^i}(u, v) = \int_{\Omega^i} a \frac{\partial u}{\partial x} \frac{\partial v}{\partial x} + b \frac{\partial u}{\partial y} \frac{\partial v}{\partial y} dx dy \quad (5)$$

and  $u$  and  $v$  range over all shape functions in  $\Omega^i$ . The local contribution  $s_i$  to  $s$  is determined similarly from  $\{F(v)\}$ . For our study, we do not form  $S$  or  $s$  explicitly, but work directly with the local versions of  $S_i$  and  $s_i$ .

2. *Condensation of the local stiffness matrices.* The local stiffness matrices and right hand sides have the form

$$S_i = \begin{bmatrix} A_i & B_i \\ B_i^T & C_i \end{bmatrix}, \quad s_i = \begin{pmatrix} b_i \\ c_i \end{pmatrix}. \quad (6)$$

$A_i$  corresponds to interactions among internal shape functions,  $C_i$  corresponds to interactions among side and nodal shape functions, and  $B_i$  corresponds to interactions between internal shape functions and side and nodal shape functions. Before solving for the unknowns associated with the frame of the partitioning  $\mathcal{P}$ , the unknowns associated with the interior  $\Omega^i$  can be decoupled from the system. This process of *condensation*, or elimination of internal unknowns, entails computing the Schur complement

$$\tilde{C}_i = C_i - B_i^T A_i^{-1} B_i, \quad (7)$$

and modifying the right hand side in a similar manner:

$$\tilde{c}_i = c_i - B_i^T A_i^{-1} b_i. \quad (8)$$

We will also normalize these quantities so that all local diagonal matrix entries are one, i.e.

$$\hat{C}_i = D_i^{-1/2} \tilde{C}_i D_i^{-1/2}, \quad \hat{c}_i = D_i^{-1/2} \tilde{c}_i, \quad (9)$$

where  $D_i = \text{diag}(\tilde{C}_i)$ . This is equivalent to scaling the shape functions.

3. *Computation of the frame unknowns.* After the internal unknowns are eliminated, the result is a system of linear equations

$$\hat{S}\hat{\alpha} = \hat{s} \quad (10)$$

for the unknowns associated with the frame of  $\mathcal{P}$ . Here

$$\hat{S} = \sum_i \hat{C}_i, \quad \hat{s} = \sum_i \hat{c}_i,$$

and  $\hat{C}_i$  and  $\hat{c}_i$  are determined from (9) and (8). We solve (10) using the preconditioned conjugate gradient method (PCG) [22]. For the preconditioner, we use the submatrix of  $\hat{S}$  associated with *nodal* unknowns. That is, if the entries of  $\hat{S}$  are arranged in the form

$$\begin{bmatrix} R & U \\ U^T & Q \end{bmatrix},$$

where  $Q$  corresponds to connections among nodal unknowns and  $R$  corresponds to connections among side unknowns, then the preconditioner for (10) is given by

$$\begin{bmatrix} I & 0 \\ 0 & Q \end{bmatrix}.$$

It is shown in [3] that the condition number of the resulting preconditioned matrix is

$$O(1 + (\log p)^2). \quad (11)$$

Hence, the number of PCG iterations required for convergence is independent of  $h$ , and it grows very slowly as a function of  $p$ .

4. *Computation of the internal unknowns.* After the unknowns  $\hat{\alpha}_i$  associated with the boundary  $\Gamma_i$  have been computed at step 3, the internal unknowns  $\alpha_i$  associated with  $\Omega^i$  can be computed by solving the system

$$A_i \alpha_i = b_i - B_i \hat{\alpha}_i,$$

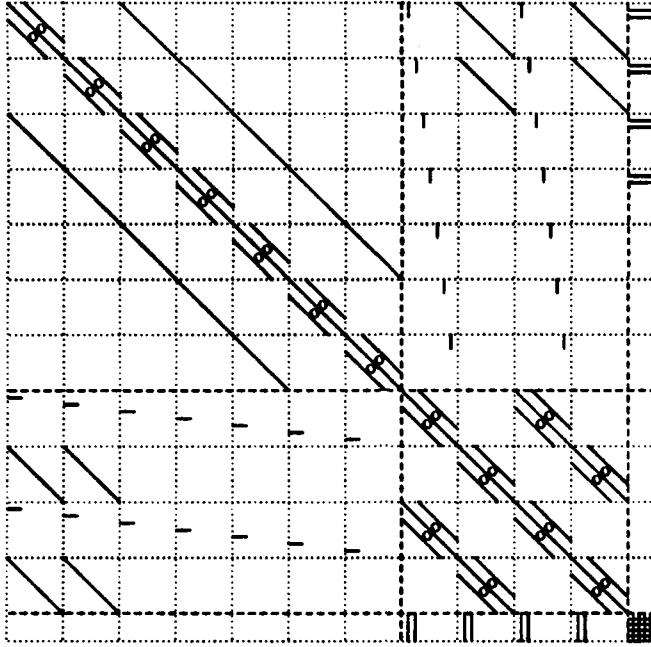


Figure 2: The nonzero structure of the local stiffness matrix for the  $Q(p)$  discretization. Solid lines indicate nonzeros, dashed lines delineate the sets  $\mathcal{I}$ ,  $\mathcal{S}$  and  $\mathcal{N}$ , dotted lines delineate blocks of size  $p - 1$ , and “o” identifies a zero band.

using the Cholesky factorization of  $A_i$  computed in step 2.

**3.2. Local Stiffness Matrix Computations.** Assume that the finite element discretization is made on an  $n \times n$  element grid, and we have a parallel architecture with  $k$  processors where  $k$  divides  $n^2$ . The first two steps of the solution algorithm, construction of the local stiffness matrices and condensation, are naturally parallelizable. There are  $n^2$  elements, so that there are  $n^2$  local stiffness matrices to be constructed, each of which contains a subblock corresponding to a set of internal unknowns to be eliminated. All of these computations are obviously independent, so that they can be executed in parallel. Each processor has  $n^2/k$  elements assigned to it, for which it constructs the associated local stiffness matrices and then performs the corresponding condensation.

Consider the construction of the local stiffness matrices. The entries of  $S_i$  (6) are determined using (5). For general operators and domains, these entries must be computed by some quadrature rule that depends on the coefficients  $a$  and  $b$ , the shape of  $\Omega^i$ , and the mapping  $T_i$  from  $\mathcal{E}$  to  $\Omega^i$ . In general, the resulting local stiffness matrix  $S_i$  is dense, although its nonzero structure may also be affected by these criteria. For rectangular elements, the shape functions specified in §2 have the form  $\theta_j(x)\theta_k(y)$  where  $\theta_j$  is a polynomial of degree  $j$ . Therefore, (5) simplifies to an expression of the form

$$B_{\Omega^i}(u, v) = \int \int a\theta'_j(x)\theta'_l(x)\theta_k(y)\theta_m(y) + b\theta_j(x)\theta_l(x)\theta'_k(y)\theta'_m(y) dx dy. \quad (12)$$

In this study, we are restricting our attention to the case where  $a$  and  $b$  are constant. For these problems, (12) further simplifies to

$$B_{\Omega^i}(u, v) = a I_x(j, k, l, m) + b I_y(j, k, l, m), \quad (13)$$

where

$$I_x(j, k, l, m) = I_1(j, l) I_0(k, m), \quad I_y(j, k, l, m) = I_0(j, l) I_1(k, m) \quad (14)$$

and

$$I_0(s, t) = \int \theta_s \theta_t, \quad I_1(s, t) = \int \theta'_s \theta'_t. \quad (15)$$

This reduces the costs of constructing  $S_i$ , since (15) can be computed in closed form, and many entries are zero. In particular, for both the  $Q(p)$  and  $Q'(p)$  discretizations,  $S_i$  has order  $\rho_i = O(p^4)$ , but only  $O(p^2)$  entries are nonzero. (See the Appendix.) For example, for the  $Q(p)$  discretization, if the rows and columns of  $S_i$  are ordered using the lexicographic ordering of shape functions

$$\begin{aligned} & \Phi_{22}^{(T)}, \Phi_{32}^{(T)}, \dots, \Phi_{p2}^{(T)}, \Phi_{23}^{(T)}, \dots, \Phi_{pp}^{(T)}, \\ & \Phi_2^{(S,1)}, \dots, \Phi_p^{(S,1)}, \Phi_2^{(S,2)}, \dots, \Phi_p^{(S,2)}, \Phi_2^{(S,3)}, \dots, \Phi_p^{(S,3)}, \Phi_2^{(S,4)}, \dots, \Phi_p^{(S,4)}, \\ & \Phi^{(N,1)}, \Phi^{(N,2)}, \Phi^{(N,3)}, \Phi^{(N,4)}, \end{aligned}$$

then the nonzero structure for the  $Q(p)$  discretization is shown in Fig. 2.

In our code for constructing the local matrices, each entry of  $S_i$  is computed using (13) – (15), where the indices  $j$ ,  $k$ ,  $l$ , and  $m$  range over all values corresponding to the lower triangle of  $S_i$ .  $I_x$ ,  $I_y$ ,  $I_0$  and  $I_1$  can be thought of as representing FORTRAN functions, and (15) is computed in closed form using (23), (27), (28) and analogues for handling side and nodal shape functions. The routines corresponding to  $I_0$  and  $I_1$  are called only if the result is nonzero, as specified by (11). Thus, the construction of  $S_i$  entails  $O(1)$  scalar computations per entry, giving a total cost of  $O(p^4)$ . Computation of a zero entry entails several queries about the indices  $j$ ,  $k$ ,  $l$  and  $m$  and at most three floating point operations; computation of a nonzero entry entails subroutine calls to  $I_x$ ,  $I_y$ ,  $I_0$  and  $I_1$ , the latter two of which require on the order of 10 scalar floating point operations.

Now consider the other purely local operation, the condensation of the local stiffness matrix, to produce the Schur complement  $\tilde{C}_i$  of (7). To perform this step, we compute the Cholesky factorization  $A_i = L_i L_i^T$ , and then compute  $\tilde{B}_i = L_i^{-1} B_i$  and  $C_i - \tilde{B}_i^T \tilde{B}_i$ . These operations were implemented using (a slightly modified version of) off-the-shelf software from the BLAS2 subroutine library, which is designed to take advantage of vector architectures [16]. A general description of the algorithm is as follows. Assume that in (6),  $S_i$  has order  $\rho$  and  $A_i$  has order  $\lambda \leq \rho$ . For any matrix  $M$  with the same dimensions as  $S_i$ , let

$$m_{\mu:\gamma,\nu} = (m_{\mu\nu}, m_{\mu+1,\nu}, \dots, m_{\gamma,\nu})^T$$

denote the column vector consisting of entries  $\mu$  through  $\gamma$  of the  $\nu$ 'th column of  $M$ , and let

$$M_\mu = \begin{cases} [m_{\sigma\tau}], \mu \leq \sigma \leq \rho, \nu \leq \tau \leq \mu - 1 & \text{if } \mu \leq \lambda \\ [m_{\sigma\tau}], \mu \leq \sigma \leq \rho, \nu \leq \tau \leq \lambda & \text{if } \mu > \lambda. \end{cases}$$

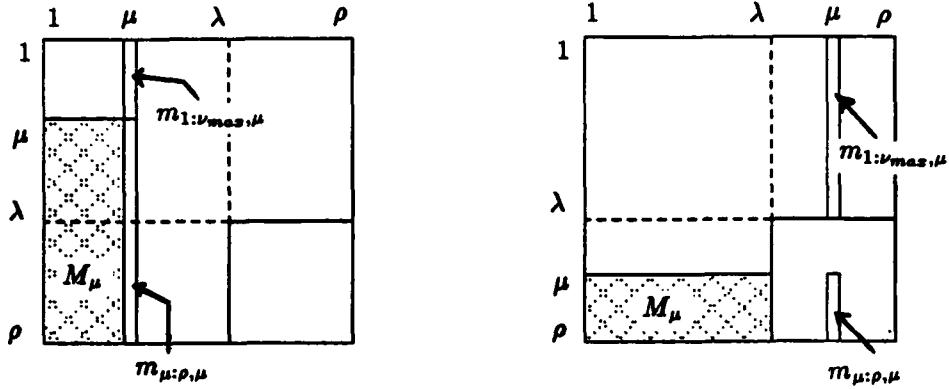


Figure 3: Submatrices and subvectors used for internal elimination. Quantities used for a Cholesky factorization step are shown on the left. Quantities used to compute the Schur complement are shown on the right.

$M_\mu$  is a submatrix of  $M$  in the lower left hand corner of  $M$ . (See Fig. 3.) In the following code fragment,  $M$  initially contains the local matrix  $S_i$  of (6); as the computation proceeds, the contents of  $M$  are dynamically modified. In steps 1 through  $\lambda$ , the lower triangle of  $A_i$  is overwritten by the Cholesky factor  $L_i$ , and  $B_i^T$  is overwritten by  $\tilde{B}_i^T$ . In steps  $\lambda + 1$  through  $\rho$ , the lower triangle of  $C_i$  is overwritten with that of  $\tilde{C}_i$ .

```

for  $\mu = 1$  to  $\rho$  do
     $v_{\max} \leftarrow \min\{\mu - 1, \lambda\}$ 
     $m_{\mu:\rho,\mu} \leftarrow m_{\mu:\rho,\mu} - M_\mu m_{1:v_{\max},\mu}$ 
    if ( $\mu \leq \lambda$ ) then
         $m_{\mu\mu} \leftarrow \sqrt{m_{\mu\mu}}$ 
         $m_{\mu+1:\rho,\mu} \leftarrow m_{\mu+1:\rho,\mu}/m_{\mu\mu}$ 
    endif
enddo

```

(16)

Except for  $m_{1:v_{\max},\mu}$ , only the lower triangle of  $M$  is referenced. In the program used to implement this computation, only the lower triangle of  $M$  is stored, by column in *packed form*. That is, the contents of the array containing  $M$  are

$$m_{1,1}, m_{2,1}, \dots, m_{\rho,1}, m_{2,2}, m_{3,2}, \dots, m_{\rho,2}, m_{3,3}, \dots, m_{\rho,\rho}.$$

(Since the vector  $m_{1:v_{\max},\mu}$  is, therefore, not available in contiguous storage,  $m_{\mu,1:v_{\max}}$  is accumulated in a temporary vector at each step of the outer ( $\mu$ ) loop.) The submatrices and subvectors of  $M$  used in one step of internal elimination are depicted graphically in Fig. 3.

The algorithm (16) is essentially the “GAXPY” form of Gaussian elimination [18, 21, 22], in which the main large-scale operation at each step is a matrix-vector product

$$m_{\mu:\rho,\mu} \leftarrow M_\mu m_{1:v_{\max},\mu}. \quad (17)$$

The computations are arranged to take advantage of architectures with vector registers, by computing (17) as a linear combination of the columns of  $M_i$ . Our implementation is essentially that

of the BLAS2 library [16].<sup>1</sup> In principle, the vector  $m_{\mu:\rho,\mu}$  can be accumulated in one or more vector registers without being stored to memory until the computation (17) is complete.

For the  $Q(p)$  and  $Q'(p)$  discretizations, this implementation of the condensation step requires  $O(p^4)$  floating point operations. This is determined by the cost of the matrix-vector product (17), and it is also strongly influenced by our choice of implementation. Consider the  $Q(p)$  discretization, where  $\lambda = (p-1)^2$  and  $\rho = (p+1)^2$ . A feature of the BLAS2 software used for (17) is that only the nonzero entries of the vector  $m_{1:\nu_{\max},\mu}$  are used for the linear combination of columns of  $M_\mu$ . There are at most 3 such nonzeros for steps 1 through  $\lambda$ , and an average of  $2.5(p-1)$  nonzeros for steps  $\lambda+1$  through  $\rho$ . At step  $\mu$ , the block  $M_\mu$  contains  $(p+1)^2 - (\mu-1)$  rows. Hence, the number of floating point multiplications performed is approximately

$$\sum_{\mu=1}^{(p-1)^2} 3[(p+1)^2 - (\mu-1)] + \sum_{\mu=(p-1)^2+1}^{(p+1)^2} 2.5(p-1)[(p+1)^2 - (\mu-1)] \approx 1.5p^4 + 26p^3.$$

These computations are vectorized, but they do not take full advantage of sparsity, since  $M_\mu$  is treated as a dense matrix. An implementation that operated only on the nonzeros of  $M_\mu$  would require  $O(p^2)$  operations.

It is evident from this discussion that many factors contribute to the cost of construction and condensation of the local stiffness matrices. As we have noted, our program takes some advantage of the special structure of the test problem, but it does not make full use of sparsity in either construction or condensation. By way of comparison, consider the situation for more general problems, where the  $O(p^4)$  entries of  $S_i$  are computed by applying a quadrature rule to (5) or, for rectangular domains, (12). Typically,  $O(p)$  quadrature points are used in both the  $x$  and  $y$  coordinates, so that (ignoring the costs of function evaluations), (5) will require  $O(p^6)$  floating point computations. For (12), this cost can be reduced to  $O(p^5)$  by taking advantage of the tensor product structure [27]. The condensation is, asymptotically, an  $O(p^6)$  computation. Of course, asymptotics also do not tell the whole story, since in general we work with relatively small values of  $p$  (on the order of 10); we expect asymptotic characterizations to be pessimistic for these values [5]. Both construction and condensation allow for significant amounts of vectorization, e.g. in the quadratures for construction and as in §3 for condensation. Our implementation is intended to take advantage of special problem structure in a “natural” way: in the matrix construction, by performing some computation for each entry, but not performing unnecessary quadratures; and in the condensation, by handling sparsity only in the manner inherited from standardized software. We believe that this implementation gives a plausible picture of the relative costs of construction and condensation; for more complex problems, absolute costs will be higher.

Step 4 of the solution algorithm, recovery of the internal unknowns, also entails purely local computations. However, because the solutions to our benchmark problems are identically zero, we did not experiment with this stage of the algorithm.

**3.3. Computation of the Frame Unknowns.** Next, consider the solution of the global linear system (10) by the preconditioned conjugate gradient method. We use the standard implementation of PCG, as described for example in [22], Algorithm 10.3.1. Each step of the iteration

---

<sup>1</sup>We used a slight modification of the subroutine DTPMV from the BLAS2 library. DTPMV computes a matrix-vector product  $w = Lv$  where  $L$  is a lower triangular matrix stored in packed form; our modification allows variations on outer loop counters to handle rectangular subblocks of  $L$ .

requires a matrix-vector product by the coefficient matrix  $\hat{S}$ , a preconditioning solve of the form  $w \leftarrow Q^{-1}v$ , and a set of *vector operations* consisting of three inner products  $\alpha \leftarrow v^T w$  and three `daxpy`'s  $w \leftarrow \alpha v_1 + v_2$ . (This is one more inner product than specified in [22]. The extra one is used for a stopping test; see §4.1.)

The preconditioning operator and vectors required by PCG are represented with global indices; implementation issues associated with these quantities are discussed in §3.4. In this section, we focus on the matrix-vector product, which shares some of the purely local character of the local stiffness matrix computations. The global matrix  $\hat{S}$  is not constructed explicitly; instead the matrix-vector product is computed as

$$w = \sum_i w_i = \sum_i \hat{C}_i v_i = \hat{S}v, \quad (18)$$

where the sum is taken over all elements, and  $\hat{C}_i$  is the Schur complement associated with an individual element. The vector  $v_i$  is gathered from a globally indexed vector  $v$ , the local matrix-vector product  $w_i = \hat{C}_i v_i$  is performed, and then  $w_i$  is used to update the global vector  $w$ . Hence, the steps required for the local matrix-vector product are:

- a. *Index and copy*: determine the indices in  $v$  corresponding to  $v_i$  and copy  $v_i$  from  $v$ .
- b. *Arithmetic*:  $w_i = \hat{C}_i v_i$ .
- c. *Update*: accumulate  $w_i$  into  $w$ .

For the parallel implementation, each processor performs this computation on all elements assigned to it. Ignoring any memory conflicts, all processors can read from the global vector  $v$  and compute the local matrix-vector product independently, so that steps (a) and (b) can be implemented with a high degree of parallelism. However, for the program to be correct, no more than one processor can write into a given location of  $w$  at any time. We enforce this by synchronizing all writes into  $w$ , so that execution of step (c) by different processors is performed serially. (See §3.4 for the method used to achieve this.) For the arithmetic step (b), recall that only the lower triangle of  $\hat{C}_i$  is stored, by column. The actual computation has the form shown in the following code fragment, in which  $\gamma = \rho - \lambda$  is the order of  $\hat{C}_i$ , and for the sake of simplicity, the subscript  $i$  is omitted:

```
for  $\mu = 1$  to  $\gamma$  do
     $w \leftarrow w + v_\mu \hat{c}_{\mu:\gamma,\mu}$ 
     $w_\mu \leftarrow w_\mu + (\hat{c}_{\mu+1:\gamma,\mu})^T \hat{c}_{\mu+1:\gamma,\mu}$ 
enddo
```

(19)

That is, multiplication by the lower triangle is done using a linear combination of the columns, and the additional inner product and accumulation for the upper triangular multiplication is performed in the same loop. No extraneous vector writes (of  $w_i$ ) or reads (of columns of  $\hat{C}_i$ ) need be performed. (Thus, this is also essentially a BLAS2 type computation [16].) For our test problems, approximately 50% of the entries of  $\hat{C}_i$  are nonzero; however, as in the condensation,  $\hat{C}_i$  is treated as a dense matrix.

**3.4. Other Coding Conventions.** We conclude this section with an outline of our coding conventions. All code was written in Alliant FX/8 FORTRAN and compiled using the global

“-O” optimization switch. All vectorizable loops were preceded with the Alliant compiler directives VECTOR and (for global inner products) ASSOC. Thus, all computations are fully vectorized. Although parallelism on the Alliant FX/8 can be achieved using compiler constructs, the compiler does not permit easy control of individual processors, and it also does not permit data driven synchronization of the type required for the matrix-vector product. To circumvent these difficulties, we implemented all the local computations described above using the scheduling program SCHEDULE [19]. For a local computation such as construction of the local stiffness matrix, SCHEDULE runs on  $k$  processors by initiating  $k$  processes consisting of  $n^2/k$  matrix constructions. This is a relatively simple use of this software which has the property that any overhead associated with it is amortized over  $n^2/k$  large-scale computations. Compiler-generated parallelism is explicitly prevented using Alliant FORTRAN compiler directives (i.e. NOCONCUR). Synchronization of the updates required by the matrix-vector products in PCG is enforced using SCHEDULE’s *lockon* and *lockoff* primitives, which force processes to spin-wait when access to  $w$  is restricted. Finally, to handle SCHEDULE’s requirement that multiple copies of subroutines be used simultaneously, all compilation was done with the “-recursive” switch.

Computations not discussed in detail above are the construction and factorization of the preconditioner, and the preconditioning and vector operations (inner products and scalar-vector products) performed during the PCG iteration. All these computations are *global* operations, in the sense that they are concerned with global quantities associated with the nodal and side unknowns. Constructing the preconditioner consists of assembling the (global) nodal operator  $Q$  from entries of the local stiffness matrices, and then computing the Cholesky factorization  $Q = LL^T$ . The factorization was performed using band elimination [22]. The preconditioning operation consists of forward-solves  $w \leftarrow L^{-1}v$  and backsolves  $w \leftarrow L^{-T}v$ . The preconditioning and vector operations all contain a large amount of natural parallelism, but not at the element level. As a result, parallelism for these tasks was handled using Alliant compiler directives (CONCUR).

**4. Experimental Results.** In this section, we present the results of a series of numerical tests of the algorithm of §3. For several problems, we give a general overview of costs, and then we show how these costs are broken down by individual computational tasks. Our objectives are both to show how the methods perform, and to understand what aspects of algorithms and computer architecture affect performance. In particular, we examine the influences of local and global computations required by algorithms, and of architectural considerations such as number of processors, vectorization and cache memory. We remark that we are not examining the issue of accuracy of the computed solution here. Correlations between accuracy requirements and cost will be discussed in a subsequent report [4].

**4.1. Machine independent results.** For most results presented in this section, we used benchmark problems with the  $Q(p)$  discretization on  $n \times n$  grids, with  $p = 4, 8$ , and  $16$ , and  $n = 4, 8, 16$  and  $32$ . In addition, we have observed [4] that often the  $Q(p)$  and  $Q'(p)$  discretizations provide solutions of comparable accuracy when  $p_Q \approx \sqrt{2}p_{Q'}$ , so that we examined the  $Q'(p)$  discretization with  $p = 6, 11$  and  $23$ , on the same grids. The choices for degree represent moderate, large and very large values. The values  $p = 16$  for  $Q(p)$  and  $p = 23$  for  $Q'(p)$  are larger than values typically used in practice and are studied primarily to see trends in the data; these

values were not considered on the  $32 \times 32$  grid. Tables 1 and 2 show the number of global and local unknowns of various types associated with these problems.<sup>2</sup>

In all experiments, the problems were posed with  $s \equiv 0$ , so that the solution to (10) is  $\hat{\alpha} \equiv 0$ . The stopping criterion for the PCG iteration was based on the relative error in the energy norm,

$$(\hat{\alpha}^{(j)}, \hat{S}\hat{\alpha}^{(j)})^{1/2}/(\hat{\alpha}^{(0)}, \hat{S}\hat{\alpha}^{(0)})^{1/2} \leq .5 \times 10^{-3},$$

where  $\{\hat{\alpha}^{(j)}\}$  are the PCG iterates and the initial guess  $\{\hat{\alpha}^{(0)}\}$  is a vector of random numbers between  $-1$  and  $1$ . Table 3 shows the number of iterations required to reach this stopping criterion. Note that these iteration counts are consistent with condition numbers of the form (11).

**4.2. Overview of timing results.** We first give a general overview of CPU times needed to solve these benchmark problems. For all experiments, reported times are in seconds, and they represent averages over three runs. The timings were determined from the “user time” returned by the Unix function *etime*; the measurements exclude timing overhead [1]. Speedup is defined to be the ratio of CPU time using one processor to CPU time on multiple processors; the same program was used in all experiments, so that the timings on one processor include a small amount of overhead associated with the scheduler.

Tables 4 and 5 show timing statistics and speedups for the  $Q(p)$  and  $Q'(p)$  discretizations, respectively, for the entire solution procedure. Table 6 shows the efficiencies of these computations, defined to be the ratio of speedup to number of processors. These results show that, in general, speedups are higher for both larger values of  $p$  and for larger grid sizes. For the  $Q(p)$  shape functions, efficiency on 8 processors ranges from 40% for the smallest grid ( $4 \times 4$ ) and polynomial degree ( $p = 4$ ), where scheduling overhead is high; to a maximum of 85%, corresponding to maximum speedup of slightly under 7. There are some examples of slight declines in efficiency when  $p$  increases from 8 to 16. The  $Q'(p)$  shape functions incur larger costs and they have slightly higher efficiencies, but otherwise they are qualitatively similar to the results for  $Q(p)$  shape functions.

Because the  $Q(p)$  basis functions have lower costs, we restrict our attention to them in the sequel. Table 7 shows a breakdown of costs and speedups of several of the individual tasks performed by the solution algorithm, for  $n = 16$  and three values of  $p$ . This data corresponds to the third row of each block row in Table 4. The computations are broken into three large-scale steps, consisting of the construction and condensation of the local stiffness matrices, the construction and factorization of the nodal preconditioning matrix, and the preconditioned conjugate gradient iteration for computing the nodal and side unknowns. The first of these steps entails purely local computations, the second is associated with the global (nodal) mesh, and the third requires both local and global computations. We see the following trends in this data:

- The costs are dominated by local stiffness matrix computations (construction and elimination). Since these are purely local, they are very highly parallelizable, and we see speedups on eight processors of 7.47, 7.25 and 6.86, respectively, for  $p = 4, 8$  and  $16$  (efficiencies of 93%, 91% and 86%). Thus, efficiencies are generally high, although there is a decline as  $p$  increases.

---

<sup>2</sup>For simplicity of programming, we constrained the four vertices of  $\partial\Omega$  by adding a constant to the diagonal entries of  $\hat{C}$ , associated with these vertices. This corresponds to constraining the vertices by spring supports. It does not influence any results discussed below.

Table 1: Number of global unknowns for benchmark problems.

		$\mathcal{Q}(p)$			$\mathcal{Q}'(p)$		
		Internal	Frame	Total	Internal	Frame	Total
$p = 4 (\mathcal{Q})$	$4 \times 4$ grid	144	145	289	96	225	321
	$8 \times 8$ grid	576	513	1089	384	801	1185
	$16 \times 16$ grid	2304	1921	4225	1536	3009	4545
	$32 \times 32$ grid	9216	7425	16641	6144	11649	17793
$p = 6 (\mathcal{Q}')$	$4 \times 4$ grid	784	305	1089	576	425	1001
	$8 \times 8$ grid	3136	1089	4225	2304	1521	3825
	$16 \times 16$ grid	12544	4097	16641	9216	5729	14945
	$32 \times 32$ grid	50176	15873	66049	36864	22209	59073
$p = 16 (\mathcal{Q})$	$4 \times 4$ grid	3600	625	4225	3360	905	265
	$8 \times 8$ grid	14400	2241	16641	13440	3249	16689
	$16 \times 16$ grid	57600	8449	66049	53760	12257	66017

Table 2: Number of unknowns in each element for benchmark problems.

	$\mathcal{Q}(p)$			$\mathcal{Q}'(p)$			
	Internal	Frame	Total	Internal	Frame	Total	
$p = 4$	9	16	25	$p = 6$	6	24	30
$p = 8$	49	32	81	$p = 11$	36	44	80
$p = 16$	225	64	289	$p = 23$	210	92	302

Table 3: Iteration counts for benchmark problems.

	$\mathcal{Q}(p)$			$\mathcal{Q}'(p)$		
	$p = 4$	$p = 8$	$p = 16$	$p = 6$	$p = 11$	$p = 23$
$4 \times 4$	16	20	27	13	17	24
$8 \times 8$	15	19	26	13	17	20
$16 \times 16$	14	18	23	13	16	20
$32 \times 32$	14	18	-	12	16	-

Table 4: Timings and speedups for  $Q$ -type shape functions.

		Timings				Speedups			
		Number of processors				Number of processors			
		1	4	6	8	1	4	6	8
$p = 4$	$4 \times 4$ grid	0.707	0.273	0.244	0.223	1.00	2.59	2.90	3.17
	$8 \times 8$ grid	2.552	0.795	0.623	0.533	1.00	3.21	4.09	4.79
	$16 \times 16$ grid	9.935	2.874	2.134	1.767	1.00	3.46	4.66	5.62
	$32 \times 32$ grid	41.564	11.533	8.556	7.071	1.00	3.60	4.86	5.88
$p = 8$	$4 \times 4$ grid	3.764	1.095	0.884	0.685	1.00	3.44	4.26	5.50
	$8 \times 8$ grid	14.652	3.955	2.900	2.259	1.00	3.70	5.05	6.49
	$16 \times 16$ grid	57.984	15.389	10.795	8.598	1.00	3.77	5.37	6.74
	$32 \times 32$ grid	234.367	61.726	43.171	34.611	1.00	3.80	5.43	6.77
$p = 16$	$4 \times 4$ grid	37.186	10.089	7.926	5.826	1.00	3.69	4.69	6.38
	$8 \times 8$ grid	147.922	40.207	28.984	22.372	1.00	3.68	5.10	6.61
	$16 \times 16$ grid	587.828	156.811	111.230	87.065	1.00	3.75	5.29	6.75

Table 5: Timings and speedups for  $Q'(p)$  shape functions.

		Timings				Speedups			
		Number of processors				Number of processors			
		1	4	6	8	1	4	6	8
$p = 6$	$4 \times 4$ grid	0.919	0.314	0.272	0.237	1.00	2.93	3.38	3.89
	$8 \times 8$ grid	3.488	1.019	0.778	0.643	1.00	3.42	4.48	5.43
	$16 \times 16$ grid	13.906	3.886	2.822	2.287	1.00	3.58	4.93	6.08
	$32 \times 32$ grid	56.100	15.223	11.084	9.019	1.00	3.69	5.06	6.22
$p = 11$	$4 \times 4$ grid	4.232	1.192	0.959	0.736	1.00	3.55	4.41	5.75
	$8 \times 8$ grid	16.684	4.462	3.230	2.524	1.00	3.74	5.17	6.61
	$16 \times 16$ grid	65.974	17.387	12.312	9.625	1.00	3.79	5.36	6.86
	$32 \times 32$ grid	266.086	69.846	48.806	38.806	1.00	3.81	5.45	6.86
$p = 23$	$4 \times 4$ grid	50.465	13.661	10.678	7.761	1.00	3.69	4.73	6.50
	$8 \times 8$ grid	201.691	53.934	38.800	30.163	1.00	3.74	5.20	6.69
	$16 \times 16$ grid	796.017	211.542	148.829	117.944	1.00	3.76	5.35	6.75

Table 6: Overall efficiency.

		$Q(p)$			$Q'(p)$		
		Processors			Processors		
		4	6	8	4	6	8
$p = 4 (Q)$	$4 \times 4$ grid	65%	48%	40%	73%	56%	49%
	$8 \times 8$ grid	80	68	60	86	75	68
	$16 \times 16$ grid	87	78	70	90	82	76
	$32 \times 32$ grid	90	81	74	92	84	78
$p = 6 (Q')$	$4 \times 4$ grid	86	71	69	89	74	72
	$8 \times 8$ grid	93	84	81	94	86	83
	$16 \times 16$ grid	94	90	84	95	89	86
	$32 \times 32$ grid	95	91	85	95	91	86
$p = 8 (Q)$	$4 \times 4$ grid	92	78	80	92	79	81
	$8 \times 8$ grid	92	85	83	94	82	84
	$16 \times 16$ grid	94	88	84	94	89	84

Table 7: Breakdown of timing costs and speedups for  $Q(p)$  shape functions on a  $16 \times 16$  grid.

		Timings				Speedups			
		Number of processors				Number of processors			
$p = 4$	1	4	6	8	1	4	6	8	
	Construct / condense LSM	6.265	1.614	1.104	0.839	1.00	3.88	5.67	7.47
	Construct / factor precon.	0.452	0.127	0.117	0.114	1.00	3.55	3.86	3.96
	PCG iteration	3.219	1.132	0.912	0.815	1.00	2.84	3.53	3.95
	Complete computation	9.935	2.874	2.134	1.767	1.00	3.46	4.66	5.62
$p = 8$	1	4	6	8	1	4	6	8	
	Construct / condense LSM	48.594	12.520	8.578	6.703	1.00	3.88	5.66	7.25
	Construct / factor precon.	0.497	0.141	0.127	0.121	1.00	3.52	3.91	4.12
	PCG iteration	8.893	2.727	2.089	1.774	1.00	3.26	4.26	5.01
	Complete computation	57.984	15.389	10.795	8.598	1.00	3.77	5.37	6.74
$p = 16$	1	4	6	8	1	4	6	8	
	Construct / condense LSM	555.791	147.678	104.142	81.037	1.00	3.76	5.34	6.86
	Construct / factor precon.	0.671	0.192	0.172	0.159	1.00	3.49	3.91	4.22
	PCG iteration	31.366	8.940	6.917	5.869	1.00	3.51	4.54	5.34
	Complete computation	587.828	156.811	111.230	87.065	1.00	3.75	5.29	6.75

- The construction and factorization of the (nodal) preconditioning matrix represents a small percentage of the overall cost.
- The PCG iteration also represents a small percentage of the computation, although it is more costly than the construction of the preconditioner. The speedups achieved for this part of the computation are smaller than those of the local matrix computations, but they are strictly increasing as  $p$  increases.

**Remark 1** The preconditioning operations and the CG vector operations were implemented in parallel using compiler directives, and we did not limit the number of processors on which these computations were performed. As a result, the timings for 4 and 6 processors are underestimates. However, as we will show below, the contributions of both these operations to overall cost is small, so that these timings do give a reasonable picture of performance.

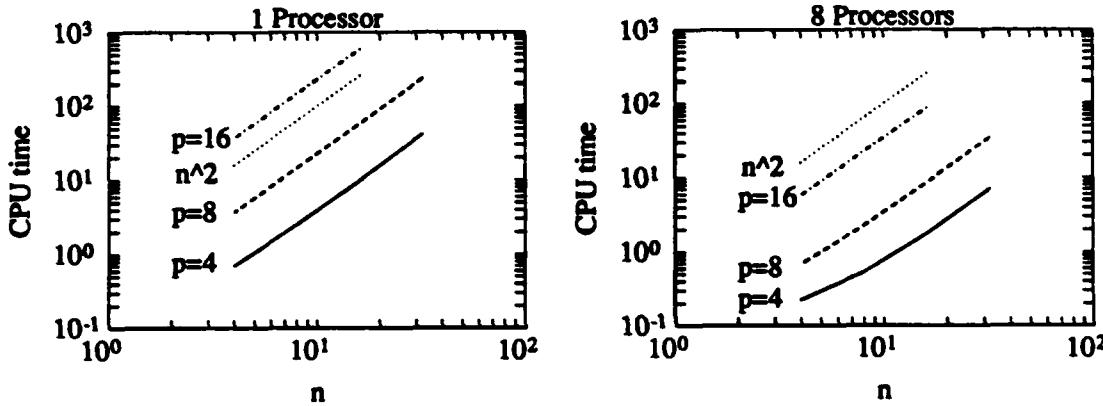


Figure 4: CPU times as functions of  $n$ , on loglog scale.

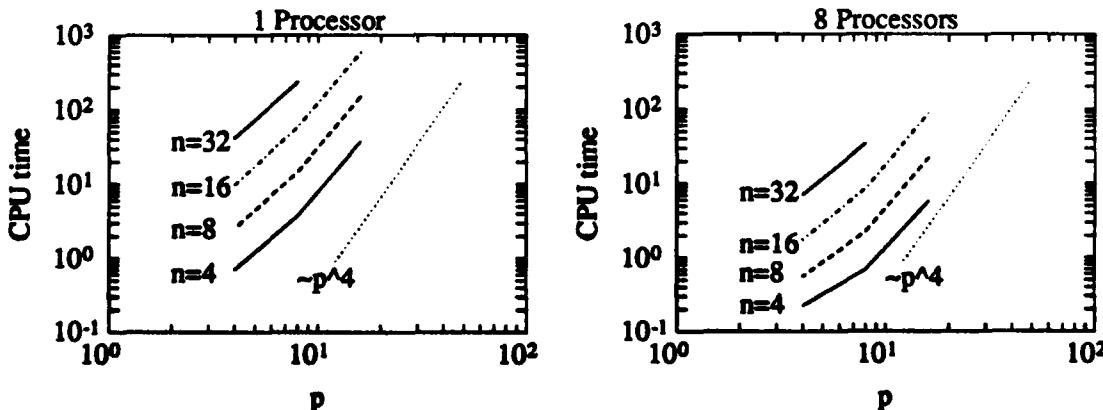


Figure 5: CPU times as functions of  $p$ , on loglog scale.

Figs. 4 and 5 show the asymptotic behavior of the timings from Table 4, as functions of  $n$  and  $p$  respectively, in loglog scale. Both figures reflect the fact that costs are dominated by the local matrix computations. Thus, for any fixed  $p$ , there are  $n^2$  independent local constructions and condensations, and the costs grow like  $n^2$ . For fixed  $n$  and large  $p$ , growth is slightly slower

Table 8: Breakdown of timing costs of local stiffness matrix computations on one processor, for a  $16 \times 16$  grid.

	$p = 4$	$p = 8$	$p = 16$
Construct	3.975	33.242	366.280
Condense	2.581	15.056	184.986
Total	6.556	48.298	551.266

than  $O(p^4)$ , the asymptotic cost; for example, the line segment between  $p = 8$  and  $p = 16$ , for one processor and  $n = 16$ , has slope 3.34. For small  $p$ , growth is closer to  $O(p^{2.5})$  because of the larger cost of computing the  $O(p^2)$  nonzero entries.

**4.3. Refined breakdown of costs.** We now refine and elaborate on the timing results of Tables 4 – 7, showing how subsidiary steps of the tasks represented in Table 7 compare in cost. For these refined statistics, we compute costs on a single processor and supplement these results with discussion of how synchronization and memory conflicts affect performance on multiple processors.

First, consider the local stiffness matrix computations, i.e. construction and condensation of the local matrices. Table 8 shows the CPU times for each of these two steps on one processor, for  $n = 16$  and  $p = 4, 8$  and  $16$ .<sup>3</sup> The results indicate that construction of the local matrices is more expensive than condensation. Since the matrix construction often involves a less regular set of computations than the condensation, we expect this phenomenon to be more pronounced for more general problems.

As noted above (see Table 7), although the local stiffness matrix computations corresponding to different elements are independent of one another, parallel efficiency declines as  $p$  increases. From Table 7, it is evident that efficiency also goes down as the number of processors grows. Fig. 6 shows a more detailed picture of these phenomena. The curves represent speedups of the local matrix computations on four and eight processors, for  $n = 8, 12$  and  $16$ , and  $p = 4$  through 18 in increments of 2. (As above, each curve represents average CPU times over three runs.) Here, the two sets of curves in each part of the figure correspond to two versions of the condensation step. The first version is the one used for all experiments described thus far; as shown in §3, it takes some advantage of sparsity of the local matrices. The second version takes no advantage of sparsity during the condensation, so that the computation (7) is performed as though all participating matrices are dense.<sup>4</sup> Thus, this version is considerably more expensive. The same procedure for constructing the local matrices, as described in §3, was used with both versions of the condensation. The results of these figures show that there is indeed a decline in speedup as  $p$  increases, especially for the more costly version of condensation; in addition, efficiency is greater on four processors than on eight processors.

To understand these issues, it is necessary to examine the computer architecture in more detail. A feature of the Alliant FX/8 is that data moves between main memory and compu-

<sup>3</sup>To prevent calls to the timer from affecting measurements, the data in Tables 8 and 9 was generated separately from that in Tables 4 and 7, and Table 10 was produced separately from all of these. This is why these tables do not contain identical subtotals for identical computations.

<sup>4</sup>This entails removing checks for zero entries in the vector  $m_{1:n_{\text{max}}, p}$  for the matrix-vector product (17).

tational elements (i.e. processors) through a cache memory. Main memory and cache memory are connected by two buses, and cache memory and computational elements are connected by a crossbar switch with four paths to the cache [1], [17], [26]. Thus, there are two sources of delay associated with movement of data between memory and processors: (i) it will take longer for data to move between processors and main memory than between processors and cache memory; (ii) when multiple processors are used, there will be contention for the buses (between main and cache memories), and possibly some delay in moving data through the crossbar switch (between cache memory and processors). We expect the crossbar switch to be less of a bottleneck than the buses [26]. However, if more than two processors attempt to move data to or from main memory, some processors will have to wait for access to the buses. Consequently, contention for the buses will tend to increase any delays caused by the need to move data between main and cache memories.

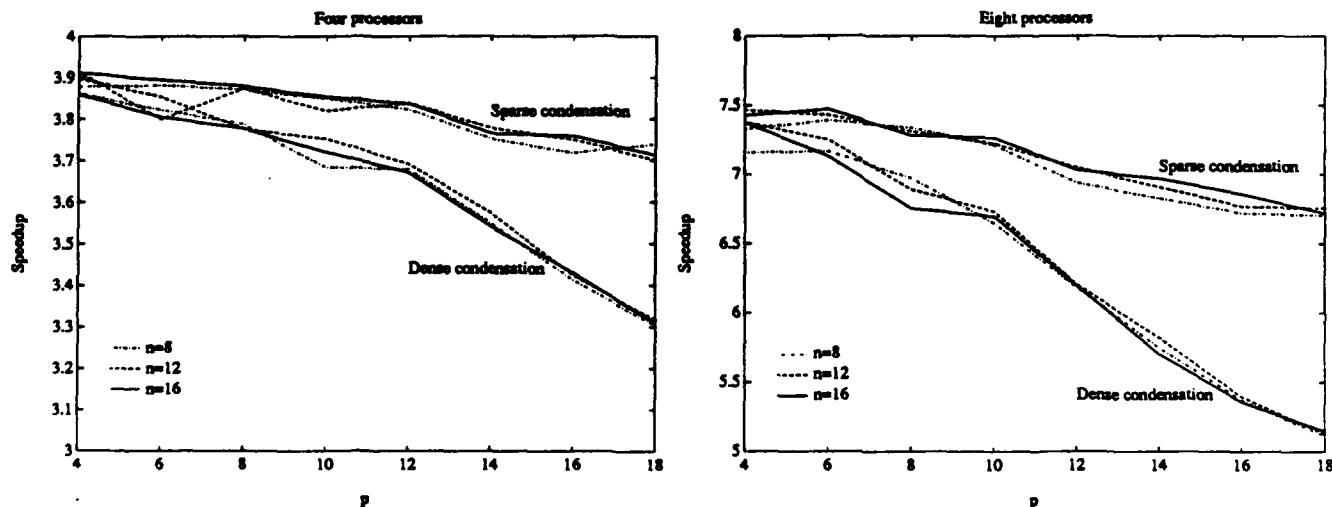


Figure 6: Speedups of local stiffness matrix computations for three meshes, on four and eight processors.

Although it is difficult to prove rigorously that these observations provide a complete explanation of parallel performance, the results of Fig. 6 are consistent with them. The machine used for these experiments has a cache memory with 512K bytes, or 64K double precision words. For the local stiffness matrix computations, each processor works with one block of storage of size equal to the number of entries in the upper triangle of the local stiffness matrix (approximately  $(p + 1)^4/2$ ). Consequently,  $k$  such blocks of storage fit into cache provided

$$k \times (p + 1)^4/2 \leq 65536,$$

which gives  $p \leq 12$  for  $k = 4$  and  $p \leq 10$  for  $k = 8$ . Examination of Fig. 6 shows steeper declines in speedups at precisely these values of  $p$ . The delays are greater for dense condensation, which we attribute to the additional memory references required for this version. We suspect that the (less pronounced) drops in speedup for smaller problems are partly explained by the lesser delays

associated with the crossbar switch. In addition, in general we have no control over how data is distributed between cache memory and main memory, and it may be that the cache is not used with maximum efficiency even when all local matrices could fit into it. Thus, there may be some additional congestion between the two levels of memory even for smaller problems. Finally, note that Fig. 6 suggests that speedup is essentially unaffected by the number of elements. This is substantiated by the following simple analysis. Let  $c_s$  denote the cost of processing a single local stiffness matrix in a serial computation, and assume that the parallel cost is increased to  $c_p = (1 + \delta)c_s$ , where  $\delta$  (which may increase with  $k$ ) reflects the delay due to memory contention. Then the speedup on  $k$  processors is

$$\frac{n^2 c_s}{n^2 / (k c_p)} = k / (1 + \delta),$$

i.e. it is independent of the number of elements.

**Remark 2** The last observation contrasts with our previous statement that speedups are somewhat lower for very coarse grids, i.e.  $n = 4$ , especially when  $p$  is also small. (See Tables 4 and 5.) For such small problems, smaller speedups are the result of scheduling overhead, which is amortized over a relatively small amount of computation.

**Remark 3** The effects of memory conflicts of the type discussed above can also be diminished by implementing the condensation with BLAS3-type constructs [15], which are designed to use cache memory efficiently.

The PCG iteration does not contribute as much to overall cost as the local computations. Nevertheless, consideration of its individual steps still reveals some interesting properties of the particular form of the matrix-vector product (18), as well as some differences between global and local operations. We group the iterations into four subsidiary operations: matrix-vector products  $w \leftarrow \hat{S}v$ ; preconditioning solves  $w \leftarrow Q^{-1}v$ ; and two types of vector operations, inner products  $\alpha \leftarrow v^T w$  and daxpy's  $w \leftarrow \alpha v_1 + v_2$ , of which there are three each. Table 9 shows a breakdown of the costs of these individual operations on a  $16 \times 16$  grid, using one processor, and Table 10 further refines the details of the matrix-vector product. Here, "arithmetic" refers to the computation (19) used to perform the matrix-vector product; "index/copy" refers to the identification of global locations of local vectors and the copying of entries of global vectors ( $v$  in (18)) to local vectors ( $v_i$ ); and "synchronize/copy" refers to the copy from  $w_i$  to  $w$ , plus the execution of the synchronization functions that prevent simultaneous writes to shared locations of  $w$ . "I/O" refers to the cost of copying the local stiffness matrix from one memory location to another.<sup>5</sup>

These results indicate that the matrix-vector product dominates the PCG iteration. This is largely a consequence of floating point operation counts (multiplications and additions), which are summarized as follows:

matrix-vector product:	$32n^2 p^2$
CG-vector operations:	$24n^2 p$
preconditioning solves:	$4n^3$ .

<sup>5</sup>This cost is an artifact of the program design, which allows either in-core or out-of-core storage for local stiffness matrices, but in both cases requires that the local matrix be explicitly read from some source. This data movement could have been avoided, so that the cost of the matrix-vector is artificially high. It does not, however, alter our general conclusions. The cost of I/O would be much higher with out-of-core techniques.

Table 9: Breakdown of timing costs of local stiffness and PCG computations on one processor, for a  $16 \times 16$  grid.

	$p = 4$	$p = 8$	$p = 16$
Matrix-vector product	2.917	8.310	30.186
Precondition	0.226	0.304	0.441
Inner product	0.047	0.126	0.339
Daxpy	0.062	0.168	0.435
Total	3.252	8.908	31.401

Table 10: Breakdown of timing costs of matrix-vector product, for a  $16 \times 16$  grid.

	$p = 4$	$p = 8$	$p = 16$
Arithmetic	1.861	5.654	21.174
Index/copy	0.634	0.946	1.886
Synchronize/copy	0.296	0.670	1.373
I/O	0.262	1.210	5.449
Total	3.053	8.480	29.882

The first line of Table 11 shows the ratios of operation counts for the matrix-vector products to operation counts for the other two steps, for  $n = 16$  and several values of  $p$ ; the data reveals the dominance of the matrix-vector product. The second line of the table shows the analogous ratios of CPU times, where the timing data for the CG-vector and preconditioning operations comes from Table 9, and the data for the matrix-vector product is from the "Arithmetic" entry of Table 10. We see that the results for operation counts agree qualitatively with those for CPU times, but they do not tell the whole story. For example, in the comparison of matrix-vector product and CG-vector operations, the ratios for operation counts are smaller than those for CPU times, indicating that the CG-vector operations are implemented more efficiently. Other factors that affect performance are vector startups and vector lengths. Each of the  $n^2$  local matrix-vector products requires  $4p$  vector startups, giving a total of  $4n^2p$  startup overhead for the matrix-vector product; this contrasts with just 6 vector startups for the CG-vector operations. In addition, we are using only the lower triangle of the matrix  $\hat{C}_i$ , so that some of the vectors used in (19) are smaller than the Alliant's basic vector length of 32. Hence, although all these computations are vectorizable, performance for the matrix-vector is somewhat lower than for the CG-vector operations. The effect of startup overhead will be diminished on multiple processors, since some startups will be performed in parallel. In the comparison of matrix-vector and preconditioning steps, we see that for  $p \geq 8$ , the cost of the preconditioning (band-) solve is higher than the operation counts predict (i.e. the ratios of CPU times are smaller than the ratios of operation counts); this is because its typical vector length is the bandwidth,  $n$ , or 16 for these data, i.e. less than 32.

A second general issue, which limits the speedups achieved by the PCG computations (Table 7), is the need to synchronize the results of local matrix-vector products in forming the global product (18). Consider the following analysis. Let  $c_p$  denote the fully parallel part of the local

Table 11: Ratios of costs of PCG operations on a  $16 \times 16$  grid.

	Matrix-vector product vs. CG-vector operations			Matrix-vector product vs. Preconditioning		
	p=4	p=8	p=16	p=4	p=8	p=16
Ratio of operation counts	5.7	10.7	21.3	4.0	32.0	128.0
Ratio of CPU times	16.2	16.7	27.7	8.0	18.0	48.0

Table 12: Comparison of speedup model with actual speedups of the PCG iteration, for a  $16 \times 16$  grid.

Processors	$p = 4$		$p = 8$		$p = 16$	
	Model	Actual	Model	Actual	Model	Actual
4	3.35	2.87	3.45	3.26	3.66	3.39
6	4.32	3.50	4.56	4.30	5.07	4.65
8	5.06	4.05	5.43	5.03	6.27	5.39

matrix-vector product, consisting of the “index/copy” and “arithmetic” and “I/O” steps; here we are ignoring any memory conflicts that may exist in these steps. Let  $c_s$  denote the serial part, consisting of the “synchronize/copy” steps. The cost of the global matrix-vector product on one processor is  $n^2(c_p + c_s)$ . In a parallel computation, processes will often have to wait for access to  $w$ , and the wait can be as long as  $(k - 1)c_s$ . Suppose every local matrix-vector product waits this long. The cost of the parallel computation is then approximately

$$\frac{n^2}{k} (c_p + (k - 1)c_s),$$

so that the speedup is approximately

$$\frac{c_p + c_s}{\frac{n^2}{k} (c_p + (k - 1)c_s)} = k \left( \frac{c_p + c_s}{c_p + (k - 1)c_s} \right). \quad (20)$$

Since the matrix-vector product dominates the PCG iteration, we will also take (20) as a measure of the speedup achievable by PCG. Table 12 compares the values of (20) (where  $c_p$  and  $c_s$  are taken from Table 10) with the actual speedups from Table 7. The results suggest that (20) is a good indicator of the qualitative behavior of the PCG iteration. We attribute the fact that the accuracy of the model decreases with additional processors to added memory conflicts.

**Remark 4** Although this model is pessimistic in the sense that  $(k - 1)c_s$  may be a long waiting time, we have observed empirically that processors do not reach the synchronization step in a fixed order, so that there are large delays for many local computations. We also note that it is possible to decrease synchronization overhead using better bookkeeping techniques to identify specific locations of  $w$  that are available.

**4.4. Comments on Performance.** Finally, we discuss the performance, in terms of floating point operations, of parts of the code that are both vectorized and implemented in parallel.

Consider the condensation of the local stiffness matrices, which entail Cholesky factorization and computation of the Schur complement. To simplify operation counts, in the experiments considered here we used the dense version of condensation, as described in the discussion of Fig. 6. For the  $Q(p)$  discretization, the floating point operation (multiplications and additions) counts are:

$$\begin{aligned} \text{Factor } A_i = L_i L_i^T : & \quad \frac{1}{3}(p-1)^6 + (p-1)^4 - \frac{4}{3}(p-1)^2 \\ \text{Compute } \hat{B}_i = L_i^{-1} B_i : & \quad 4(p-1)^2((p-1)^2 + 1)p \\ \text{Compute } \hat{C}_i = C_i - \hat{B}_i^T \hat{B}_i : & \quad 4p(4p-1)(p-1)^2. \end{aligned}$$

CPU times on one processor, for  $n = 16$  and  $p = 8$  and  $16$ , are 28.4 and 874.5 seconds respectively, giving performances in millions of floating point operations per second (Mflops) of 1.55 for  $p = 8$  and 2.35 for  $p = 16$ . Based on the speedups for the local computations from Fig. 6 (6.75 for  $p = 8$  and 5.36 for  $p = 16$ ), this gives performance estimates on eight processors of 10.5 and 12.6 Mflops, respectively. Similar results were also obtained for the matrix-vector product (18) – (19), with a maximum rate of 3 Mflops on one processor (for local matrices of order approximately 500).

By way of contrast, the LINPACK benchmark (for dense elimination with dense matrices of order 100) on a single processor of an Alliant FX/4 is 2.1 Mflops [14], and on eight processors, the BLAS2 kernels with arguments in main memory achieve 18 – 20 Mflops ([21], p. 81). Hence, our performance on vectorized code is at best comparable to that of the BLAS2 kernels. Note that this is lower than performance achieved for a variety of matrix operations reported e.g. in [21], where BLAS3-type blocking strategies lead to performance of upwards of 30 Mflops. There are several reasons for this. First, despite the fact that (17) and (19) are designed to avoid unnecessary stores of the accumulating products  $m_{\mu:\rho,\nu}$  and  $w$ , examination of the generated assembler code reveals that the actual computations are not performed efficiently. For example, in principle, the outer loop of (17) requires a daxpy with one argument (columns of  $M_\mu$ ) taken from memory, but no loads or stores to memory; the actual code performs one store, one load, and a daxpy with one argument in memory. Thus, there are three times as many memory references as is necessary, leading to a degradation of performance on the order of 50%. Second, we have little control over management of the cache memory; we suspect that because there are many local matrices being processed, they are likely to be located in main memory rather than cache memory. The good performances exhibited in [21] were achieved using hand coded assembler [20]. We believe that better performance of the techniques under consideration here can be obtained using more sophisticated coding techniques. However, since these tasks do not have the dominant cost of the overall computation, further tuning will not affect our conclusions, and we have not pursued this issue. For more complex problems, e.g. where local stiffness matrices are constructed by (vectorized) quadrature, it would be imperative to implement the quadratures efficiently.

**5. Conclusions.** In this paper, we have examined the computational costs of an implementation of the  $hp$ -version of the finite element method on the Alliant FX/8, a shared memory parallel computer. Our main conclusions are as follows:

1. Costs are dominated by the local computations, i.e. construction of local stiffness matrices and condensation of these matrices for elimination of internal unknowns.

2. Global operations, particularly the preconditioning associated with nodal unknowns, contribute a relatively small amount to overall cost.
3. Communication and synchronization costs associated with the use of unassembled local stiffness matrices for CG iteration do not greatly degrade performance, and their effects are understood.
4. The likelihood of memory conflicts places limitations on the sizes and number of local problems that can be handled efficiently. We expect this problem to be ameliorated through the use of more sophisticated coding techniques such as those available in the BLAS3 [15] or LAPACK [2] libraries, but we do not expect it to disappear entirely.

Thus the “natural parallelism” associated with the decomposition of problems by elements can be exploited in a straightforward manner to get good speedups, provided the size or number of local problems are not too large. These conclusions apply to a particular type of architecture, a shared memory machine with a relatively small number of processors. We expect similar results to apply to other machines in this class, e.g. the CRAY-2.

We now discuss how we expect our observations to carry over to other classes of problems and computers.

1. *Different problem coefficients or domain topologies.* As long as the element grid is topologically rectangular, the general methodology described here should be applicable. As shown in §3, if the coefficients of (1) are more complex, or if the domain is less regular, then the fully local computations will be more expensive, and we expect these costs to be more dominant. For highly anisotropic problems (e.g. large  $a/b$ ) or discretizations with very flat rectangular elements, there may be an increase in PCG iteration counts, but we suspect this will not offset the dominance of the local computations.
2. *Use of the  $h$ -version.* If the  $h$ -version is used for discretization, then the analogue of the solution method presented here is domain decomposition, with local super-elements consisting e.g. of  $p^2$  elements. In this case, for PCG iterations to display convergence rates independent of problem size, it is necessary to perform some type of modification of the super-element side and internal shape functions [3],[6],[10]. We expect conclusions similar to those above to hold for such methodologies. An alternative for achieving fast convergence is to use standard shape functions but apply a relatively fine nodal preconditioner [24]. For such a strategy, a larger percentage of computational effort is devoted to the sparse matrix factorization and solves associated with preconditioning than we have observed.
3. *Adaptive methods.* In contrast to the methodology considered here, where all operators were computed “from scratch,” the  $hp$ -method is often implemented in a hierarchical manner, where higher order elements are used to supplement previously computed low order operators. In this case, the local computations will be somewhat less dominant. Many issues along these lines, such as load balancing if different order basis functions are used in different elements, as well as mesh refinement strategies, remain open.
4. *Shared memory computers with more processors.* Increasing the number of processors will decrease the costs of the local stiffness matrix computations more than those of the other computations. For example, for all of the problems of Table 7, we estimate that increasing the number of processors by as much as a factor of four (to thirty-two) will decrease the local costs significantly, but the effect on the costs of PCG will be small. In such a scenario, for  $p \geq 8$  local costs will still

dominate. In light of the fact that our local costs are artificially low, we expect our conclusions to apply for shared memory machines with on the order of fifty processors. However, for this to be borne out, it will be necessary for the local matrix computations to be implemented so that memory conflicts do not limit efficiency.

5. *Local memory computers.* We do not attempt to make a precise statement about this class of architectures, but outline some of the issues. The memory conflicts associated with local matrix computations on shared memory computers should not be a factor on local memory machines, so that we expect the local computations to be more efficient on the latter class of architectures. The matrix-vector products will entail exchanges of data corresponding to super-element boundaries, but we expect the effect of this overhead to be similar to that of the synchronization required by the shared memory implementation. It will be necessary to implement the global preconditioner and other CG operations efficiently.

**Appendix.** We outline the properties of the shape functions that give rise to sparse local stiffness matrices for constant coefficient problems. Consider the representation

$$S_i = \begin{bmatrix} (\mathcal{I}, \mathcal{I}) & (\mathcal{I}, \mathcal{S}) & (\mathcal{I}, \mathcal{N}) \\ (\mathcal{S}, \mathcal{I}) & (\mathcal{S}, \mathcal{S}) & (\mathcal{S}, \mathcal{N}) \\ (\mathcal{N}, \mathcal{I}) & (\mathcal{N}, \mathcal{S}) & (\mathcal{N}, \mathcal{N}) \end{bmatrix}, \quad (21)$$

where each entry represents a block matrix containing all terms (5) in which the shape functions come from the indicated set. Thus, for example, the block  $(\mathcal{S}, \mathcal{I})$  contains all terms in which  $u \in \mathcal{S}$  and  $v \in \mathcal{I}$ . (In (6),  $A_i$  corresponds to  $(\mathcal{I}, \mathcal{I})$ .) From properties of the Legendre polynomials, it can be shown that the following relations hold:

$$\begin{aligned} \text{In } (\mathcal{I}, \mathcal{I}): \quad B_{\Omega^*}(\Phi_{jk}^{(T)}, \Phi_{lm}^{(T)}) \neq 0 & \quad \text{iff} \quad j = l \text{ or } j = l \pm 2 \text{ and } k = m; \text{ or} \\ & \quad k = m \text{ or } k = m \pm 2 \text{ and } j = l. \\ \left. \begin{array}{l} B_{\Omega^*}(\Phi_j^{(S,1)}, \Phi_{lm}^{(T)}) \neq 0 \\ B_{\Omega^*}(\Phi_j^{(S,3)}, \Phi_{lm}^{(T)}) \neq 0 \end{array} \right\} & \quad \text{iff} \quad j = m \text{ and } l = 2 \text{ or } l = 3. \\ \left. \begin{array}{l} B_{\Omega^*}(\Phi_j^{(S,2)}, \Phi_{lm}^{(T)}) \neq 0 \\ B_{\Omega^*}(\Phi_j^{(S,4)}, \Phi_{lm}^{(T)}) \neq 0 \end{array} \right\} & \quad \text{iff} \quad j = l \text{ and } m = 2 \text{ or } m = 3. \\ \text{In } (\mathcal{N}, \mathcal{I}): \quad B_{\Omega^*}(\Phi^{(N,j)}, \Phi_{lm}^{(T)}) & \equiv 0. \quad (22) \\ \text{In } (\mathcal{S}, \mathcal{S}): \quad B_{\Omega^*}(\Phi_j^{(S,l)}, \Phi_k^{(S,m)}) \neq 0 & \quad \text{iff} \quad l - m \text{ is even and } j = k \text{ or } j = k \pm 2. \\ \text{In } (\mathcal{N}, \mathcal{S}): \quad B_{\Omega^*}(\Phi^{(N,k)}, \Phi_j^{(S,l)}) \neq 0 & \quad \text{iff} \quad j = 2 \text{ or } j = 3. \\ \text{In } (\mathcal{N}, \mathcal{N}): \quad B_{\Omega^*}(\Phi^{(N,j)}, \Phi^{(N,k)}) \neq 0. & \end{aligned}$$

Relations in  $(\mathcal{I}, \mathcal{S})$ ,  $(\mathcal{I}, \mathcal{N})$  and  $(\mathcal{S}, \mathcal{N})$  are determined from symmetry.

As an example of how (22) is established, consider the entries of  $(\mathcal{I}, \mathcal{I})$ . Let  $(f, g) \equiv \int_{-1}^1 f(\xi)g(\xi)d\xi$ . The Legendre polynomials satisfy [13]

$$(P_j, P_k) = \begin{cases} 1/(2j+1) & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases} \quad (23)$$

$$P_j(\xi) = \frac{1}{2j+1} (P'_{j+1}(\xi) - P'_{j-1}(\xi)) \quad (24)$$

$$P_j(-1) = (-1)^j. \quad (25)$$

From (3) and (13) – (15), we have

$$B_{\mathcal{E}}(\Phi_{jk}^{(T)}, \Phi_{lm}^{(T)}) = a(\phi_j, \phi_l)(\phi'_k, \phi'_m) + b(\phi'_j, \phi'_l)(\phi_k, \phi_m). \quad (26)$$

Consequently, (3), (24) and (25) imply that

$$\phi_j(\xi) = \frac{1}{\sqrt{2(2j-1)}} (P_j(\xi) - P_{j-2}(\xi)),$$

so that

$$(\phi_j, \phi_l) = \begin{cases} [(P_j, P_j) + (P_{j-2}, P_{j-2})] / [2(2j-1)] & \text{if } j = l \\ -(P_j, P_j) / [2\sqrt{(2j-1)(2j+3)}] & \text{if } j = l-2 \\ -(P_{j-2}, P_{j-2}) / [2\sqrt{(2j-1)(2j-5)}] & \text{if } j = l+2 \\ 0 & \text{otherwise.} \end{cases} \quad (27)$$

Moreover,  $\phi'_k(\xi) = P_{k-1}(\xi)$ , so that

$$(\phi'_k, \phi'_m) = \begin{cases} [(2k-1)/2](P_{k-1}, P_{k-1}) & \text{if } k = m \\ 0 & \text{otherwise.} \end{cases} \quad (28)$$

Thus, the first term of (26) is nonzero if and only if  $j = l$  or  $j = l \pm 2$  and  $k = m$ ; the second term is handled in an identical way.

**Acknowledgements:** We thank Kyle Gallivan for several helpful discussions.

## References

- [1] Alliant Computer Systems Corporation, Littleton, Massachusetts. *FX/FORTRAN Programmer's Handbook*, March 1987.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. Technical Report CS-90-105, Computer Science Department, University of Tennessee, 1990.
- [3] I. Babuška, A. Craig, J. Mandel, and J. Pitkäranta. Efficient preconditionings for the p-version finite element method in two dimensions. Technical Report BN-1105, Institute for Physical Science and Technology, University of Maryland, 1989.

- [4] I. Babuška and H. C. Elman. In preparation.
- [5] I. Babuška and H. C. Elman. Some aspects of parallel implementation of the finite element method on message passing architectures. *J. Comp. Appl. Math.*, 27:157–187, 1989.
- [6] I. Babuška, M. Griebel, and J. Pitkäranta. The problem of selecting the shape functions for a p-type finite element. *Int. J. Numer. Methods Engrg.*, 28:1891–1908, 1989.
- [7] I. Babuška and M. Suri. The  $p$ - and  $h\cdot p$  versions of the finite element method, an overview. *Computer Methods in Applied Mechanics and Engineering*, 80:5–26, 1990.
- [8] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, I. *Math. Comp.*, 47:103–134, 1986.
- [9] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, II. *Math. Comp.*, 49:1–16, 1987.
- [10] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, III. *Math. Comp.*, 51:415–430, 1988.
- [11] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, IV. *Math. Comp.*, 53:1–24, 1989.
- [12] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland, Amsterdam, 1978.
- [13] S. D. Conte and C. de Boor. *Elementary Numerical Analysis, An Algorithmic Approach*. McGraw-Hill, New York, 1980.
- [14] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Computer Science Department, University of Tennessee, 1989.
- [15] J. J. Dongarra, J. du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. Technical Report 86, Mathematics and Computer Science Division, Argonne National Laboratory, 1988.
- [16] J. J. Dongarra, J. du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14:1–17, 1988.
- [17] J. J. Dongarra and I. S. Duff. Advanced architecture computers. Technical Report 57, Mathematics and Computer Science Division, Argonne National Laboratory, 1989. Revision 2.
- [18] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.
- [19] J. J. Dongarra, D. C. Sorensen, K. Connolly, and J. Patterson. Programming methodology and performance issues for advanced computer architectures. *Parallel Computing*, 8:41–58, 1988.
- [20] K. A. Gallivan. Personal communication, 1990.

- [21] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32:54–135, 1990.
- [22] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, second edition, 1989.
- [23] W. D. Gropp and D. E. Keyes. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM J. Sci. Stat. Comput.*, 8:s166–s202, 1987.
- [24] W. D. Gropp and D. E. Keyes. Parallel performance of domain-decomposed preconditioned krylov methods for pdes with adaptive refinement. Technical Report RR-773, Computer Science Department, Yale University, 1990.
- [25] A. T. Patera. Advances and future directions of research on spectral methods. In A. K. Noor, editor, *Computational Mechanics: Advances and Trends, AMD-Vol. 75*, pages 411–427. ASME, New York, 1987.
- [26] Per Stenström. Reducing contention in shared-memory multiprocessors. *IEEE Computer*, 21:26–37, 1988.
- [27] A. Weiser, S. C. Eisenstat, and M. H. Schultz. On solving elliptic equations to moderate accuracy. *SIAM J. Numer. Anal.*, 17:908–929, 1980.
- [28] O. B. Widlund. Iterative substructuring methods; algorithms and theory for problems in the plane. In R. Glowinski, G. H. Golub, G. A. Meurant, and J. Periaux, editors, *Domain Decomposition Methods for Partial Differential Equations*, pages 113–128. SIAM, Philadelphia. 1978.